Specific Scheduling Support to Minimize the Reconfiguration Overhead of Dynamically Reconfigurable Hardware

Javier Resano, Daniel Mozos
Department of Computer Architecture (DACYA),
Universidad Complutense de Madrid, Spain

{javier1, mozos}@dacya.ucm.es

Diederik Verkest [1, 2, 3], Francky Catthoor [1, 2], Serge Vernalde [1]
1. IMEC vzw, Leuven, Belgium IMEC
2. Professor at Vrije Universiteit Brussel, Belgium
3. Professor at Katholieke Universiteit Leuven, Belgium
{Verkest, Catthoor, Vernalde}@imec.be

ABSTRACT

Dynamically Reconfigurable Hardware (DRHW) platforms present both flexibility and high performance. Hence, they can tackle the demanding requirements of current dynamic multimedia applications, especially for embedded systems where it is not affordable to include specific HW support for all the applications. However, DRHW reconfiguration latency represents a major drawback that can make the use of DRHW resources inefficient for highly dynamic applications. To alleviate this problem, we have developed a set of techniques that provide specific support for DRHW devices and we have integrated them into an existing multiprocessor scheduling environment. In our experiments, with actual multimedia applications, we have reduced the original overhead due to the reconfiguration latency by at least 93%.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems – *real-time and embedded systems.*

General Terms

Algorithms, Management, Performance.

Keywords

Dynamic reconfigurable hardware, run-time scheduling.

1. INTRODUCTION

Current multimedia applications, such as digital video and 3D games, present highly dynamic and non-deterministic behaviour, and a very variable workload. To cope with these features, high performance and flexibility are required. Moreover, for embedded systems the amount of resources is highly constrained and, at the same time, the number of applications that they have to support is constantly increasing. Hence, it is infeasible to provide Application Specific Integrated Circuits (ASICs) for all of them, but on the other hand, Instruction Set Processors (ISPs) cannot always provide the required performance. Hence, they need some hardware support. Dynamically Reconfigurable Hardware (DRHW) presents the ideal features to solve this problem, since it provides both high performance and run-time flexibility. Thus, its functionality can be updated at run-time to meet the variable requirements of the running applications. In addition, current commercial DRHW platforms (like FPGAs) have recently included interesting new features such as partial reconfiguration capabilities and support for IP design.

In order to take advantage of the DRHW features, dynamic task allocation support and scheduling support are needed. To tackle the dynamic task allocation to the DRHW resources, we have adopted an Interconnection Network (ICN) model for the DRHW [4]. This model provides not only task allocation support, but also inter-task communication support (including HW/SW communications), and operating system support. With this model DRHW resources can be easily integrated in a heterogeneous multiprocessor system since it presents a uniform view both for the HW and SW resources. On top of this model, any existing scheduler for multiprocessors systems can be applied. We have selected an existing hybrid run-time/design-time scheduling methodology called Task Concurrency Management (TCM) [9] since it provides run-time flexibility and, at the same time, it generates only a small run-time penalty due to its execution because most of the exploration and computation is done at design time. This methodology attempts to reduce the energy consumption of the platform while meeting the real-time constraints of the applications. Working together, the ICN model and the TCM scheduling methodology provide the desirable support to use DRHW in embedded systems. However, the run-time flexibility of DRHW often comes at the price of a very large reconfiguration overhead. For instance, reconfiguring one tenth of a Virtex XC2V6000 requires at least 4 ms. This overhead is not always acceptable for highly dynamic applications, since they may demand reconfigurations every few milliseconds. Moreover, multiprocessor schedulers for embedded systems often neglect this overhead because, typically, a task can be loaded on a SW processor in just a few microseconds as long as the task is stored in its local memory. Hence, in order to tackle this large reconfiguration overhead, DRHW resources need specific scheduling support. The goal of our work is to drastically reduce this reconfiguration overhead, making effective the use of partial reconfiguration on DRHW resources even for those applications that demand frequent partial reconfigurations. To this end, we have developed two different techniques, namely the prefetch-scheduling technique and the replacement technique. The prefetch-scheduling technique receives as input a set of tasks that must be loaded and decides when they are going to be loaded, attempting to hide the loading latency and taking into account the inter-task dependencies. The replacement technique increases the possibilities of reusing those subtasks that are more critical for the system performance. We have integrated our techniques in the TCM environment, and tested them with actual multimedia applications including a highly dynamic 3D rendering application where the reconfiguration overhead is reduced by at least 93%.

The rest of the paper is organized as follows; the next section introduces the related work; section 3 presents our DRHW model; section 4 explains our run-time scheduling flow; section 5 and 6 describe the replacement technique and the inter-task prefetch technique; section 7 presents some experimental results and finally section 8 summarizes our conclusions.

2. RELATED WORK

Previously, other research groups have addressed the minimization of the reconfiguration overhead. Much of this work proposes the development of new types of architectures, like multi-context FPGAs and especially coarse-grain architectures [2]. However, the reconfigurable market is still being clearly dominated by the FPGAs, which are becoming more and more broadly used in industry. For this reason we have selected FPGAs to analyze the effectivity of our approach although it can be equally applied to other architectures.

A very interesting approach to reduce the reconfiguration overhead for FPGAs is found in the Ph. D. thesis of Zhiyuang Li [11] where three techniques are proposed, namely configuration compression, caching and prefetching. The first technique compresses the configuration bits of a task to reduce their loading latency. This technique is orthogonal with our approach. The second technique, deals with the problem of allocating tasks in the FPGA, trying to maximize their reuse. However, they assume that a task can be placed anywhere in the FPGA which is not a realistic assumption unless a very costly run-time routing process would be performed each time that a new task is loaded. Finally, the configuration prefetching technique attempts to hide the latency of the load of a configuration by accomplishing this load before it is needed. To this end, the next task to be executed is predicted based on past events and profiled data. If the prediction is a success, it is possible to hide at least partially its reconfiguration latency; otherwise an erroneous configuration is loaded with the consequent penalization. Although we share the idea of applying a prefetch technique to reduce the reconfiguration overhead, we are proposing a totally different approach where the reconfiguration schedule is decided at run-time after analyzing the output of a run-time scheduler. Our approach presents several advantages. First, it allows reducing the computational overhead, since all the prefetch decisions for a whole graph are taken at once and most of the computation is done at design-time. Second, it prevents prediction misses, since our heuristic receives information about the subtasks scheduled in the near future. Finally, it reduces the overall execution time of the system, since our scheduling heuristic is aware of how its prefetch decisions affect the system performance and it uses this information to minimize the execution time.

Other good approaches regarding how to minimize the influence of the reconfiguration overhead applying scheduling techniques at design-time are found in [7] and [3]. However, they do not include any run-time component. Therefore, they are not suitable for dynamic applications.

3. THE ICN MODEL FOR DRHW

Our approach relies on the ICN model to provide the support for run-time allocation of tasks on the DRHW. This model is explained in detail in [4, 5]. The basic idea of the ICN model is that the platform is split into a set of identical tiles. Each tile is wrapped by a communication interface. These tiles can communicate among them using message-passing primitives over a network-on-a-chip. To support the communications there are routing tables allocated inside the communication interface. Each tile can hold one subtask at a time. When a new subtask is loaded the routing tables are updated. Thus, the other subtasks can communicate with it. One of the key points of the ICN DRHW model is that since all the tiles have identical interfaces, and these interfaces are known in advance, the routing process can be fully performed at design time avoiding a costly run-time routing process. In addition, this model provides basic operating system primitives that allow, for instance, stopping or restarting a task. This model has been successfully implemented on Virtex, and Virtex-II FPGAs. As it is shown in figure 1, the ICN model provides a SW-like view of the FPGA. Thus, an FPGA-based platform can be considered as a multiprocessor system where subtasks are assigned to FPGA tiles instead of to ISPs.

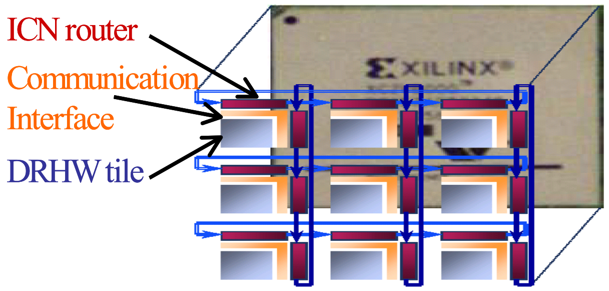


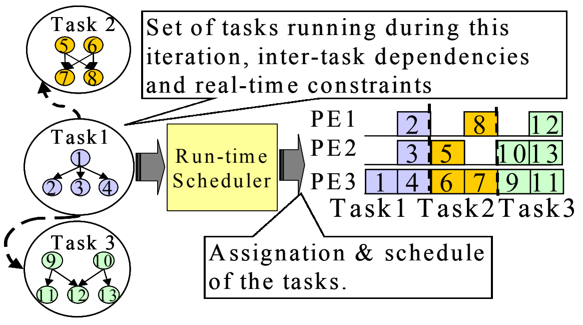
Figure 1. ICN model for DRHW (with nine tiles).

Another interesting feature of this model is that it is suitable for heterogeneous systems with both DRHW tiles and ISPs. In fact, at least one ISP must be always in the system, since it must execute the operating system and perform the task schedule. A HW/SW communication interface has been developed to support HW/SW communications using the same message-passing paradigm. A codesign environment, called OCAPI-XL, is used to develop applications for such a heterogeneous platform [8].

4. INTERACTION WITH THE TASK SCHEDULER

On top of this model, any existing scheduler for multiprocessors systems can be applied. Typically, these schedulers are not aware of the reconfiguration overhead. Hence, we have developed three modules that at run-time update the output of the scheduler, taking into account the impact of the reconfiguration overhead on the system performance. Figure 2 illustrates what we expect from this scheduler. Basically, it must be able to know which are the tasks that must be executed and to provide an assignment of the subtasks over the processing elements (PE) as well as a feasible schedule. Each task is represented using a subtask graph, which includes control and data dependencies among the subtasks and real-time constraints. The scheduling process must generate the minimum possible run-time penalty. Hence, most of the computation must be performed at design time. However, since we are targeting highly dynamic applications, we assume that part of the scheduling process must be accomplished at run-time.

We assume that an application can be described as a set of tasks (where each task is represented as a subtask graph) that interact dynamically among them. Thus, the non-deterministic behaviour must remain outside the boundaries of the tasks. If the behavior of a task depends on external data, different versions (graphs) of the same task are generated. Each of these versions is called a scenario. Thus, the idea of scenario allows supporting data dependencies and while loops inside the tasks. The run-time scheduler must tackle the dynamic behaviour of the application and select a schedule for the proper scenario of each running task



Task 2
5
6
7
8
Set of tasks running during this iteration, inter-task dependencies and real-time constraints
Task1
1
2
3
4
Run-time Scheduler
PE1
PE2
PE3
2
8
12
3
5
10
13
1
4
6
7
9
11
Task1 Task2 Task3
Task 3
9
10
11
12
13
Assignation & schedule of the tasks.

Figure 2. Run-time scheduler inputs and outputs.

To demonstrate our modules, we have integrated them into an existing hybrid run-time/design-time scheduling methodology (called Task Concurrency Management, TCM, methodology) [9]. This methodology attempts to reduce the energy consumption of the platform while meeting the real-time constraints of the applications. TCM carries out the scheduling process in two phases. First, at design-time, a Pareto curve is generated for each scenario of a task. A Pareto curve is a set of solutions where each solution is better than all the others in at least one of the parameters to optimize. An example of a Pareto curve is depicted in figure 3. This Pareto curve corresponds to a motion JPEG video decoder application, which runs in a platform with an FPGA and a SA-1110 processor. Thus, different assignments and schedules of the subtasks over this two PEs lead to different energy/performance trade-offs. The second phase is carried out at run-time. During the execution, a run-time scheduler is called periodically. It has to identify the current scenario for each running task and selects the most suitable Pareto point for them. The scheduler attempts to select the Pareto point that consumes less energy but still meets all the timing constraints of the application.

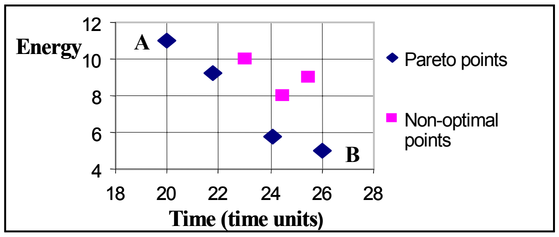


Figure 3. Pareto curve of the motion JPEG video decoder.

Thus, in the case of the example of figure 3, if there is a 20 time units timing constraint for the execution of this task, the scheduler selects point A. However, if the constraint is 30 units point B is selected with the consequent energy savings.

However, current TCM schedulers do not take into account the specific reconfiguration overhead of the FPGA tiles. This overhead not only can move the Pareto curve to a more time and energy consuming area, but it can also change the shape of the curve. We have provided support to tackle this specific overhead by including our techniques in the TCM run-time scheduling flow as is depicted in Figure 4. Once the run-time scheduler generates its schedule, it is parsed in order to generate some initial data structures with information needed for the following steps. Afterwards, three main decisions are taken. Firstly, for each task the reuse module decides which subtasks can be reused from a previous iteration. Secondly, if some of the subtasks cannot be reused, the prefetch module schedules their loads attempting to minimize the execution time overhead. Finally, each time that a new subtask is loaded, the replacement module decides to which tile it is going to be assigned.

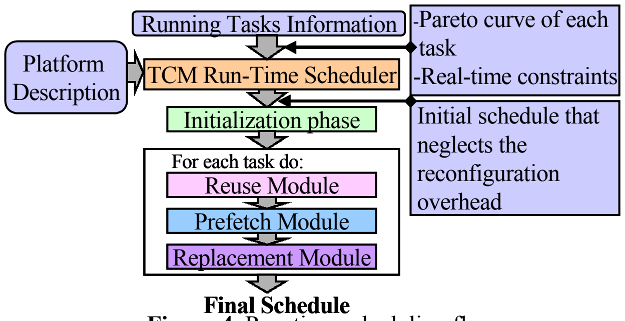


Figure 4. Run-time scheduling flow

Since our modules, apart from applying a prefetch-scheduling approach, also attempt to reuse previously loaded configurations, they require some flexibility regarding the assignment of a task on to the DRHW. For instance, a subtask that was loaded in last iteration on to tile 2 may be assigned in the next iteration to the tile 1 preventing any possibility of reuse. In order to solve this problem the run-time scheduler works with virtual addresses for the DRHW tiles. Hence, it generates a schedule where subtasks are assigned to virtual tiles. Afterwards, the reuse module and the replacement module identify the virtual tiles with the physical tiles. The ICN model supports this approach. Firstly, because it has a symmetric structure where all the tiles are identical and secondly, because simply by updating the routing tables all the communications are guaranteed.

The scheduling and replacement decisions are taken sequentially for all the tasks following the order of the initial schedule. The goal of the prefetch module is to schedule the load of a set of configurations minimizing the loading latency. Since current

FPGAs do not support simultaneous reconfigurations, only one reconfiguration can be carried out at a time. The prefetch module schedules these reconfigurations as soon as possible applying a heuristic based on list scheduling. This heuristic is explained in detail in [6]. Basically it attempts to overlap the latency of a reconfiguration with the computation of previous subtasks. If this is possible this reconfiguration does not penalize the system performance.

5. REPLACEMENT TECHNIQUE

Multimedia applications are commonly composed of recurring tasks. Hence, it is possible to reuse some tasks (or some subtasks of a task) among different iterations. From the point of view of the replacement techniques, a DRHW resource with the ICN model is equivalent to a physical memory, with subtasks instead of memory pages (with the exception that in our case the latency is much bigger). Hence, in order to minimize the number of reconfigurations, we have developed a replacement heuristic based on a well-known memory-page replacement strategy (LFD). When a new task must be loaded, LFD replaces the page that is going to be requested farthest in the future. LFD has been proven to be an optimal memory-page replacement strategy [1] when the sequence of accesses to the memory blocks is known in advance. In our case, we do not have information about the entire future since we are targeting dynamic applications. However, after parsing the initial schedule selected by the run-time scheduler, we do have some information about the near future, since the output of the scheduler is a sequence of scheduled subtasks. Hence, we can use this information to reduce the number of reconfigurations by using this sequence when applying the LFD replacement strategy. After the run-time scheduler selects its schedule, an initialisation module parses it. The goal is to identify which of the subtasks that are currently located in some of the FPGA tiles can be reused. With this information the tiles of the FPGA are divided in two categories, namely, non-reusable tiles (those that hold a subtask that is not going to be executed in the FPGA during the current scheduled sequence of tasks) and reusable tiles (those that hold a subtask that has been scheduled for execution in the FPGA). This module generates two lists as output. The first list contains all the non-reusable tiles, whereas the second contains the re-usable tiles sorted by their execution start time. This list is called the replacement list.

Each time that a subtask is going to be loaded on to the FPGA, the replacement module decides where must be loaded. There are two possibilities. First, if the non-reusable list is not empty, the first tile of the list is selected and afterwards it is removed from this list and added to the replacement list. Otherwise, the module selects one of the tiles in the replacement list. In this case, the module applies our LFD-based replacement heuristic to select the subtask to replace from the replacement list. In this list three categories exist. Firstly a set of tiles with a subtask that is going to be executed in the current iteration. Secondly, a set of tiles that have been already reused during the current period (and that are not going to be reused again during this period, otherwise they still belong to the first category), and finally, a set of tiles with the subtasks that have been loaded during this iteration. The LFD strategy gives more priority to remain in the FPGA to those subtasks that belong to the first category.

This initial replacement technique considers all the subtasks as equally important. However, some subtasks are more critical for the system execution than others. Hence, we have modified this technique to give more priority to these critical subtasks (called CNs). To this end, two replacement lists are used instead of one. One list contains the tiles with critical subtasks and the other the non-critical. In order to find which are the critical subtasks at design time we follow the pseudo-code depicted in figure 5. This code detects those subtasks which reconfiguration latency cannot be hidden by our scheduling technique. If these critical subtasks are reused it is guaranteed that no time overhead will be introduced due to the load of the other subtasks (since its reconfiguration latency can be hidden by our scheduling heuristic).

```
For each task scenario do:
  Apply the local scheduling technique;
  Mark as CN the first reconfiguration
  that penalizes the system performance;
  While (there is still overhead)
     Apply local scheduling without
     considering the overhead due to the CNs;
     Mark as CN the first reconfiguration
     that penalizes the system performance;
```

Figure 5. Pseudo code for the selection of the critical subtasks.

6. INTER-TASK PREFETCH TECHNIQUE

In [6] we have applied our prefetch heuristic separately to each task. However, since we have information about a sequence of tasks, it is also possible to apply it just one time to the whole sequence. Nevertheless, the complexity of the heuristic is O(Nl*log(Nl)) where Nl is the number of subtasks that must be loaded. Hence, the larger the number of subtasks to load, the more computational time is needed to execute our scheduling heuristic. Figure 6 depicts the execution time needed to schedule 192 subtasks loadings with our heuristic, when these subtasks are grouped in tasks of different sizes. In addition, it also depicts the reconfiguration overhead for the different sizes. The figure shows that scheduling 192 loadings in just one graph requires five times more computational time than scheduling the same number of loadings in 32 graphs of 6 subtasks per graph. On the other hand, the bigger the graphs, the better the results, since they provide more flexibility for the prefetch-scheduling process. Since this technique must be applied at run-time, combining all the subtask graphs into just one may generate an excessive run-time penalty. However, clearly, important reductions of the reconfiguration overhead can be achieved when inter-task prefetch is applied.

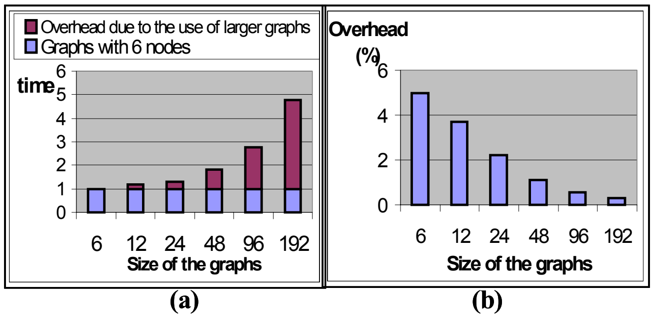


Figure 6. **(a)** Normalized execution time needed to schedule 192 subtasks grouped in graphs of different sizes. **(b)** Impact of the reconfiguration overhead on the resulting schedules.

Using again the idea of critical nodes, we have found a way to reduce the reconfiguration overhead without increasing the run-time penalty due to the execution of our heuristic. Basically, when all the loads of a graph have been scheduled, we check if there is still time to load more subtasks. In this case, we will attempt to load the CNs of the subsequent task. This modification does not significantly increase the execution time of our heuristic, since on average the number of loads to carry out for each task remains unchanged. However, as it will be shown later, it leads to very important reconfiguration-overhead reductions.

7. EXPERIMENTAL RESULTS

Since this is in part a run-time scheduling approach, our major concern is to create the minimum possible run-time overhead while providing good solutions. In this context, we have developed modules that generate a very small run-time penalty. When analysing the execution time of our modules, we have observed that the execution time of the prefetch module is much more time consuming than the others. However, it still remains low enough to be applied at run-time. The execution time of this module running on a processor at 200MHz is 4µs for a graph with 13 subtasks to be loaded. If there are 20 task graphs to be scheduled with this module, the overhead will be 0.08ms, and the overhead of all the modules together will be below 0.1ms. In order to assess the total run-time overhead, the execution time of the run-time scheduler must be considered as well. The current TCM run-time scheduler [10] provides a near-optimal schedule for a set of 20 tasks in less than 0.1 ms. Hence, for 20 tasks the entire run-time scheduling process can be executed in less than 0.2 ms., which is still affordable, especially when it is compared with the reconfiguration overhead of loading a new task on to an FPGA tile (in our system we assume that this overhead is 4 ms). We believe that this time overhead is acceptable, since our approach can lead to large time-savings due to the reuse of configurations and the near-optimal schedule of the FPGA reconfigurations.

Table 1. Set of multimedia benchmarks.

Set of Task	Sub-tasks	Ideal Ex.Time	Overhead	Prefetch
Pattern Rec.	6	94 ms	+17%	+4%
JPEG dec.	4	81 ms	+20%	+5%
Parallel JPEG	8	57 ms	+35%	+7%
MPEG encoder	5	33 ms	+56%	+18%

We have carried out two experiments to analyze the results of our techniques. In both experiments we assume that the time needed to load a subtask onto a DRHW tile is 4ms (this is the time needed to reconfigure one tenth of a Virtex XC2V6000), and that all the subtasks are executed in the DRHW resources. This is a worst-case assumption since in a heterogeneous platform some of the subtasks would be assigned to other resources. Hence, they would not introduce this 4ms overhead.

In order to compare our current approach with the approach presented in [6] we have applied our techniques to the same set of multimedia tasks used there. These tasks are a sequential and a parallel version of the JPEG decoder, an MPEG encoder, and a Pattern Recognition application that applies the Hough transform over a matrix of pixels in order to look for geometrical figures. In table 1 the features of these tasks are presented. "Ideal Ex. Time" is the execution time of the application when there is no reconfiguration overhead. "Overhead" is the percentage of the initial execution time that is added when the entire set of subtasks must be loaded on to the DRHW. Finally, "Prefetch" is the same overhead when our scheduling prefetch heuristic is applied. For the MPEG encoder there are three different scenarios corresponding to the decoding of B, P, and I frames (the table includes the average data). The appropriate scenario is selected at run-time following the sequence of frames. We have simulated 1000 iterations of the execution of this set of applications for different number of tiles. In order to introduce more dynamic behaviour, the applications executed during each iteration vary randomly. Figure 7 depicts the results of this simulation. In this figure, the *random* represents the results obtained when using the approach presented in [6], where the replacement policy was almost random. *Belady* presents the results when our LFD-based replacement strategy is applied. *Cooperative* presents the results when the CNs have higher priority in the replacement strategy. In these three cases, the results have been obtained applying the scheduling-prefetch technique separately to each task. Finally *Cooperative2* represents the results when inter-task prefetch is also applied. We called our replacement strategy cooperative because its goal is not only to reduce the number of reconfigurations but also to help the scheduling-prefetch technique to achieve the best possible results.

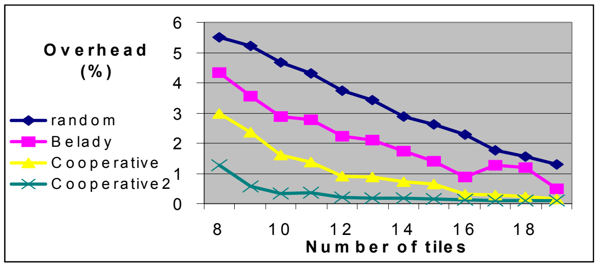


Figure 7. Reconfiguration overhead for the set of 4 tasks depicted in table 1 and a variable number of DRHW tiles, when running with a random dynamic behavior.

It must be remarked that the reconfiguration overhead virtually disappears (without optimisations the reconfiguration overhead was 23%) when our modules are active. Thus, when all our techniques are applied (in *Cooperative2*) we reduce the overhead from 23% to 1.2%. This is a reduction of 95%. Moreover, we have achieved this reduction when just 8 tiles are present. Since we are executing 23 different subtasks on the DRHW, with just 8 tiles the percentage of reused tasks is very small (for cooperative2, it is just 24%). As the number of tiles grows, the same happens with the percentages of reuse, leading to even further reductions. In figure 7 there is a result that must be explained. Surprisingly, for the LFD-based strategy, the reconfiguration overhead grows when the number of tiles goes from 16 to 17. This does not mean that the heuristic works worse with 17 tiles than with 16 since the goal of this heuristic is to reduce the number of reconfigurations and in this case, it is reduced from 22% (for 16 tiles) to 20% (for 17). However, as it has been explained previously, not all the reconfigurations generate the same overhead. Hence, it is perfectly possible that even when the number of reconfigurations decreases the reconfiguration overhead increases. This cannot happen to our cooperative heuristic, since it is aware of the impact of each reconfiguration.

As a second experiment we have tested our modules with a highly dynamic 3D rendering application. This application is composed of 6 dynamic tasks that have in total 10 subtasks. For each task several scenarios can be selected at run-time. The amount of scenarios depends on the dynamism of the task. Thus, task 5 has four scenarios, whereas task 4 has ten. In total there are 40 different scenarios. However, due to the inter-task dependencies, at run-time just 20 feasible combinations exits, which are called inter-task scenarios. The run-time scheduler does the selection among the inter-task scenarios. The average execution time of a subtask in this application is 5.7ms, which is comparable with the 4ms needed to load a subtask onto a DRHW tile. Moreover this execution time heavily varies, going from 0.2 ms to 30ms. Figure 8 depicts the results of applying our technique to this application. In this case, the reconfiguration overhead was initially 71% of the ideal execution time. Applying the approach presented in [6] it is reduced to 25%. When our new techniques are also applied, the overhead is reduced to just 5% when there are just 5 tiles. Thus, our techniques have eliminated 93% of the initial overhead. Moreover, this remaining overhead is not constant. In fact, when there are 5 tiles, 48% of the overhead is created in 3 of the 20 inter-task scenarios, and with 6 tiles, the same scenarios generate 52% of the whole overhead. The reason is that these scenarios have a very small computational load. Thus, the average execution time for a subtask in these scenarios is just 1.1 ms, i.e. four times less than the time needed to load a subtask. Hence, when a subtask must be loaded in these scenarios, our scheduling technique does not receive enough flexibility to hide the entire overhead. However, if we assign to the application a fixed timing constraint that represents the frames per second requirements, the more critical scenarios are those that require more execution time. Fortunately, our approach works very appropriately in these scenarios, since the greater the execution time, the more possibilities exist to hide the reconfiguration latency. For instance, in the five inter-task scenarios with more execution time, the overhead generated is just 0.001%.

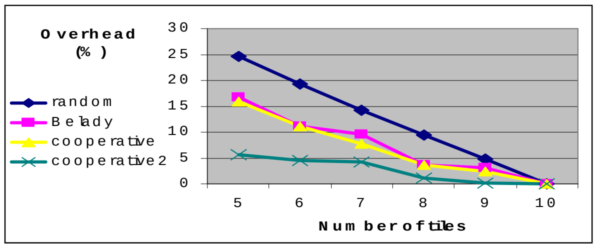


Figure 8. Reconfiguration overhead for a Pocket GL 3D rendering application, for different number of DRHW tiles.

8. CONCLUSIONS

The reconfiguration latency of current DRHW commercial platforms creates huge execution-time overheads. This fact has prevented the use of DRHW resources for highly dynamic applications, where reconfigurations are demanded every few milliseconds. With our work, we aim to demonstrate that, with the appropriate support, this drawback can be overcome and, as a consequence, DRHW can play an important role to tackle the dynamism of current multimedia applications especially for embedded systems, where is not feasible to provide ASIC support for each different application.

To test our techniques we have integrated them into an existing scheduling environment for heterogeneous multiprocessor systems, using the support provided by the ICN DRHW model. In this environment our techniques have eliminated at least 93% of the initial overhead. In addition we have observed that the remaining overhead penalizes more the scenarios with less computational load. Since we are targeting multimedia applications, which are typically frame-based and have constant Quality-of-Service (QoS) requirements, the scenarios with less computational load are also the less critical ones. In fact, a coherent task assignment policy is to just execute in the DRHW resources the scenarios with more computational load, since they need to be accelerated in order to meet the QoS requirements. In these scenarios our techniques virtually suppress the entire initial overhead (the other scenarios can be executed on a low-power ISP which is normally more energy efficient). Hence, our modules allow taking advantage of the partial reconfiguration capabilities of current DRHW resources with an affordable reconfiguration overhead. In addition, our modules achieve good results while still generating a very small run-time penalty due to their execution. Hence, they can be executed at run-time to deal with the dynamism of current multimedia applications

9. ACKNOWLEDGMENTS

The authors would like to acknowledge all our colleagues from the T-Recs and Matador groups at IMEC. This work has been partially supported by TIC 2002-00160.

10. REFERENCES

[1] Belady, L.A.,"A Study of Replacement Algorithms for Virtual Storage Computers" In IBM Systems Journal, 5, pp. 78-101, 1966.

[2] Hartenstein, R. "A decade of reconfigurable computing: A visionary retrospective". Proc. DATE, 2001. pp. 642-649, Munich, Germany, 2001.

[3] Maestre, R. et al, "Configuration Management in Multi-Context Reconfigurable Systems",ISSS'00, pp.107-113, 2000

[4] Marescaux, T. et al., "Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs", Proc. of FPL'02, pp. 795-805, 2002.

[5] Marescaux, T. et al., "Networks on chips as Hardware Components of an OS for Reconfigurable Systems", Proc. of FPL'03, p.595-605, 2003.

[6] Resano, Javier et al, "Run-Time Minimization of Reconfiguration Overhead in Dynamically Reconfigurable Systems", FPL'03, LNCS 2778, pp. 585-594, 2003.

[7] Shang, Li et al., "Hw/Sw Co-synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs", ASP-DAC'02, pp. 345-360, 2002.

[8] Vanmeerbeeck, G. et al, "Hardware/Software Partitioning for Embedded Systems in OCAPI-XL" CODES'01, 2001.

[9] Yang, Peng et al., "Energy-Aware Runtime Scheduling for Embedded-Multiprocessors SOCs", IEEE Journal on Design&Test of Computers, pp. 46-58, 2001.

[10] Yang, Peng et al. "Pareto-Optimization-Based Run-Time Task Scheduling for Embedded Systems". Proc. of ISSS'03. 2003. p.120-125.

[11] Zhiyuan Li, "Configuration management techniques for reconfigurable computing" Ph.D. thesis, ISBN: 0-493-65106-3. 2002